Samuel Tai Meng Yao
32025068

Assignment 2 Report

Table of Contents

Samuel Tai Meng Yao
32025068

Design of the code

High level description of approach

To improve the readability of the code, I have organized the codes in a systematic way in the following order:

1. All imported functions and helper functions will be at the top
2. Church encoding of the operators
3. Church encoding functions (take input and apply the operator and the inputs)
4. Component parsers of each exercise will be located before the exercise.

In the development process of this assignment, I have chosen to follow the procedures defined below for each of the exercises,

1. read and understand the requirements
2. develop BNF grammar
3. write the codes
4. evaluate and optimize the code and BNF grammar

The reason behind it is that I found out that with designed BNF grammar, I can develop the initial code for the exercises faster and clearer as I have a blueprint for me to follow. After writing the codes, I will check and try to optimize the codes I have written in terms of modularity, maintainability, efficiency, etc as this is important in Functional Programming. This step is needed because the BNF grammar that I developed earlier does not mean everything is perfect and it might contain some redundancy or flaw which needs to be avoided.

Design choices

Part	Description
1	• longLambdaP will handle only lambda expression start from bracket. • shortLambdaP is the same as lambdaP which will handle everything. • No space is allowed
2	• A general calculator will be used for all 3 of the exercises in part 2 to reduce the redundancy of redeclaring the chaining between the operators. All of them will be able to accept complexCalcP input as permitted by the teaching team.

High level structure of code

I have created many small modular parser and parser combinators in the assignment. I decide to create a new parser or parser combinator when I saw there are too much parsing needed in a single function, I will choose to separate them into different parser. This is to ensure maintainability and reusability of the code. For example, instead of parsing everything in a longLambdaP, I break it down into parsing of argument and function body, then group them up in the main parser.

Code architecture choices

Some of the code architecture choices that I have made is that I used do notation which is the syntactic sugar for bind for all the codes that I have developed that required bind. The reason I decided to do that is because it gives a better visualisation and easier to understand and implement as I do not need to chain everything together using the “>>=” symbol. Besides, I also used where

clause in some part of the codes. When there is a lengthy code that is only needed in a function, I chose to use where clause to hold the code into a variable and used it later. This is to provide a cleaner look to my code and remove the repetitive code as well if we need to use it multiple times in a single function. For example, I used where clause for the arithmetic parser that needs to handle the bracket (precedence) instead of code everything directly under it as that codes will only be used under that function and to make the code looks cleaner and more understandable.

Parsing

BNF Grammar

<lambdaP> ::= <shortLambdaP> | <longLambdaP>

<shortLambdaP> ::= <shortLambdaPAux> <shortLambdaP>?

<longLambdaP> ::= <longLambdaPAux> <longLambdaP>?

<shortLambdaPAux> ::= ("(" <shortArgumentList> <shortBodyExpr>+ ")") | (<shortArgumentList> <shortBodyExpr>+) | "(" <shortLambdaPAux> ")"

<longLambdaPAux> ::= "(" <longArgumentList> <longBodyExpr>+ ")" | "(" <longLambdaPAux> ")"

<shortBodyExpr> ::= <body> | <shortLambdaPAux>

<longBodyExpr> ::= <body> | <longLambdaPAux>

<shortArgumentList> ::= "λ" <var>+ "."

<longArgumentList> ::= "λ" <var> "."

<body> ::= "(" <var>+ ")" | <var>

<var> ::= [a-z]

Usage of parser combinator

In my implementation, the parser combinators are mainly used in the part 2 when we need to handle the precedence of the operators such as arithmetic, logical and inequality operators. Besides, I also used parser combinator such as list and list1 in my code to handle the repeated parsing of the parser that I passed into. Furthermore, there is also a heavy use of alternative parser that represented by the "|||" symbol to generate a parser that able to handle different parsing scenarios until one succeeds. There parser combinators play an important role in the code implementation as they allowed me to perform operation on the input and output another parsing function, and I would not need to worry about things will break during the execution of the code.

Choices made in creating parser and parser combinators

The choices I made when creating parser is that I will always ensure that the parser will not be too large and is task focusing. This is to ensure that every parsing function is readable, understandable and can be reusable. For example, in the longLambdaP parsing function, I choose to create another 2 parsing functions for argument and body functions, with this choice, the code is more readable, and the parsing of body function can also be reused in the shortLambdaP parsing function. Besides,

when developing parser combinators, I always choose to make it a generic datatype function to reduce code duplication.

How parsers and parsers combinators were constructed using the Functor, Applicative, and Monad typeclasses

Please refer to fmap, apply and bind section.

Functional Programming

Small modular functions

Small modular functions are heavily used in my implementation of the assignment. This is because modular function is easier to read, write, refactor, reuse, or even collaborate with different components of the codes. For example, I choose to break the church encoding of all the operators into their individual function first and then only call them in the function when needed. Consequently, as an example, I can shorten the church encoding of the “not” by substituting the church encoding of true when building the church encoding for “not” which clearly shown the benefit of small modular functions.

Composing small function together

The main parsers of the assignment are built by composing small functions together. Thus, it results in a massive usage of composite functions. The reason I developed the codes using composite function is because of the reusability and maintainability. By combining the pure functions together, we can form a more complex function and debug the small function if anything goes wrong. Hence, it can save our time and effort to maintain the code. For instance, I have combined the operator parsers, natural numbers parser and more to form a complex arithmetic parser and this arithmetic parser is used in another part of the code to form another complex functionality.

Declarative style (including point free style)

Declarative programming style is widely used in my code implementation. This is because declarative style is easier to read as it describes what the program wants instead of how it performs the function. Declarative style can be seen in most or all the codes implemented such as shortArgumentList parser and shortBodyExpr parser is used in parsing the shortLambdaP parser which describe that I want to parse the short form argument and the short form function body without giving the information about how it is being implemented. Besides, declarative approach minimizes data mutability, reusability, sharable and at the same time still retain the flexibility to implement new methods when needed. I also covert the code to point free code when the code is simple to remove the naming variables and code at the right level of abstraction.

Haskell Language Features Used

Typeclasses and Custom Types

Typeclasses such as monad, functor and applicative and more are widely used in the codes as the main components of the assignment is built around Parser, Lambda and Builder which uses typeclasses. The reason that typeclasses is used in the code implementation is because it provides ah hoc polymorphism such that a function does not need to be defined the same way for each type. This gives us the flexibility to customize how each function works with different types under that typeclasses. Moreover, by using typeclasses, we can add new types for the functions in a nonintrusive fashion which encourage the maintainability and extensibility of the code as well.

Samuel Tai Meng Yao
32025068

fmap, apply, bind

The reason fmap is useful is because that it provides the method for us to map the function to a data in a context, fmap makes our code more generic and reusable for different data type or data structure of type functor. For example, we can map some function to the content of the Parser and returning a Parser type with mapped content inside without worrying it will break the type of the code. Furthermore, bind is very useful as it allows us to pass the value in a monad context as an argument into a function that takes in that data type and return the result in the monad instance. It is especially useful when combined with the do notation which allows us to parse the string and at the same time, assigned the result of parsing to the variable on the left-hand side of the arrow which allows me to perform operations on the result later. Since Parser is an instance of functor, applicative and monad typeclasses, it provides the ability to parse something and extract the result using bind, using fmap to map the function to the value in context and using pure from applicative typeclasses to return the final value in a Parser. Therefore, it results in a chain of parser and parser combinators without worrying things will break due to different type.

Higher order functions

I did used quite some higher order functions in the assignment such as chain. This is because higher order function provide simplicity to the code and allowing the people to understand the code at a high level easily. To add on, higher order functions make it less time-consuming to write fully functional and clear code. For example, by having chain as a higher order function which takes in other function in the context of Parser enables code customisability and reusability as we can generate different functions depending on the function passed in and we able to spend less time to reimplement the almost similar function again.

Function composition

Please refer to the functional programming section where I explain on how and why I used function composition in the assignment.

Leveraging built-in functions

I always tried to use the built-in functions to perform the functionality I desired before actually creating a new function. This is because it is a waste of time and effort to create something that has already existed and by using the built-in function, we are able to code in a more declarative style as we can just call the built-in function that describes what is the operation while working on the data without showing the implementation details.

Extension

Fibonacci and Factorial

I implemented the Fibonacci and factorial both in a normal recursion way instead of using the Y-combinator. The reason behind this is I do not really understand how Y-combinator can be built and I feels like the current approach does not result in any big downside. The interesting of part of these 2 new features is that they are implemented purely using church encoding and do not used any Int type in the calculation while recursing into itself again. I found out that with the church encoding approach, the evaluation of the result takes a longer time as the calculation is getting complex and did not use the simplest form of church encoding to represent the number.

The challenging part for building these functions is that since everything is done in church encoding form, it is kind of different in how we usually check the base case as I cannot directly check whether

the value return is equal to some number such as 0. Besides, even if use the lamToBool and lamToInt function, they returned the result in a context form which makes it harder for me to compare. Hence, to solve this problem, I decided to create a new function that handle the comparison of number in church encoding form to check whether the recursion hits the base case and pattern matching the result.

Foldr

I implemented the foldr which works for the list of church encoding form that we implemented earlier in part 3. The foldr works as the usual foldr but bracket is not needed for parsing the binary function that it is going to use in the aggregation process. The interesting part of this foldr is that it can works with the data type that we defined in part 2, such as logic expression, arithmetic expression, and the combination of both. Besides, I made it such that the second and third argument of the foldr which supposed to be single value and list can be any expressions that evaluate to the desired type. This provides more flexibility and variety in using the foldr.

The complex or difficult part of the new feature is that it is hard to code out the folding process. This is because you will need to develop a solution that keep recurse to the end first before start applying the function to the input and the order of applying is also important as well. Besides, same as the previous 2 features, handle the base case of the recursion is the hardest as the foldr will not stop like usual when it hits the empty list as it would not know whether is has already empty or not. Hence, I have come out with a function to check whether the list is empty and return a Boolean value to act as the guideline for the recursion to stop.